# Downloading the Dataset

```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
```

```
X, y = fetch_openml('mnist_784', version=1, return_X_y=True, as_frame=False)
y = y.astype(int)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/datasets/_openml.py:968: FutureWa
  warn(
```

```
y = np.where(y > 4, 1, 0)
```

# Normalization on the training and test data

```
X = X.reshape((X.shape[0], -1))
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

```
X = np.concatenate([X, np.ones((X.shape[0], 1))], axis=1)
# Split the data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=10000,random_state=4
```

```
# Initialize the neural net and weights
d = 785
k = 100
std_W = np.sqrt(1 / (d))
std_v = np.sqrt(1 / (k))
W = np.random.normal(0, std_W, (k, d))
v = np.random.normal(0, std_v, k)
```

```
print(len(X_train))
print(len(y_train))
```

```
60000
60000
```

## Functions and their derivates

```
def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return np.where(x > 0, 1, 0)

def quadratic_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def quadratic_loss_derivative(y_true, y_pred):
    return 2* (y_pred - y_true)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def logistic_loss(y_true, y_pred):
    y_pred = sigmoid(y_pred)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

def logistic_loss_derivative(y_true, y_pred):
    y_pred = sigmoid(y_pred)
    return y_pred - y_true
```

```
learning_rate = 0.0001
batch_size = 10
epochs = 10
reg_lambda = 0.001
```

## Linear classifier

```
for epoch in range(epochs):
    X_train, y_train = shuffle(X_train, y_train)
    for i in range(0, X_train.shape[0], batch_size):
        X_batch = X_train[i:i+batch_size]
        y_batch = y_train[i:i+batch_size]
        # Forward pass
        h = relu(np.dot(X_batch, W.T))
        y_pred = np.dot(h, v)
        y_pred_binary = np.where(y_pred > 0, 1, 0)

        # Backward pass
        delta = (y_pred_binary - y_batch)
```

```
        grad_v = np.dot(delta, h) /batch_size + reg_lambda * v
        grad_W = np.dot((np.outer(delta, v) * relu_derivative(h)).T, X_batch) / batch_

        # Update weights
        v -= learning_rate * grad_v
        W -= learning_rate * grad_W
# Test the model
h_test = relu(np.dot(X_val, W.T))
y_test_pred = np.dot(h_test, v)
y_test_pred_binary = np.where(y_test_pred > 0, 1, 0)
print(f"Test accuracy: {np.mean(y_test_pred_binary == y_val)}")
```

```
    Test accuracy: 0.8987
```

## Neural network classifier with quadratic loss

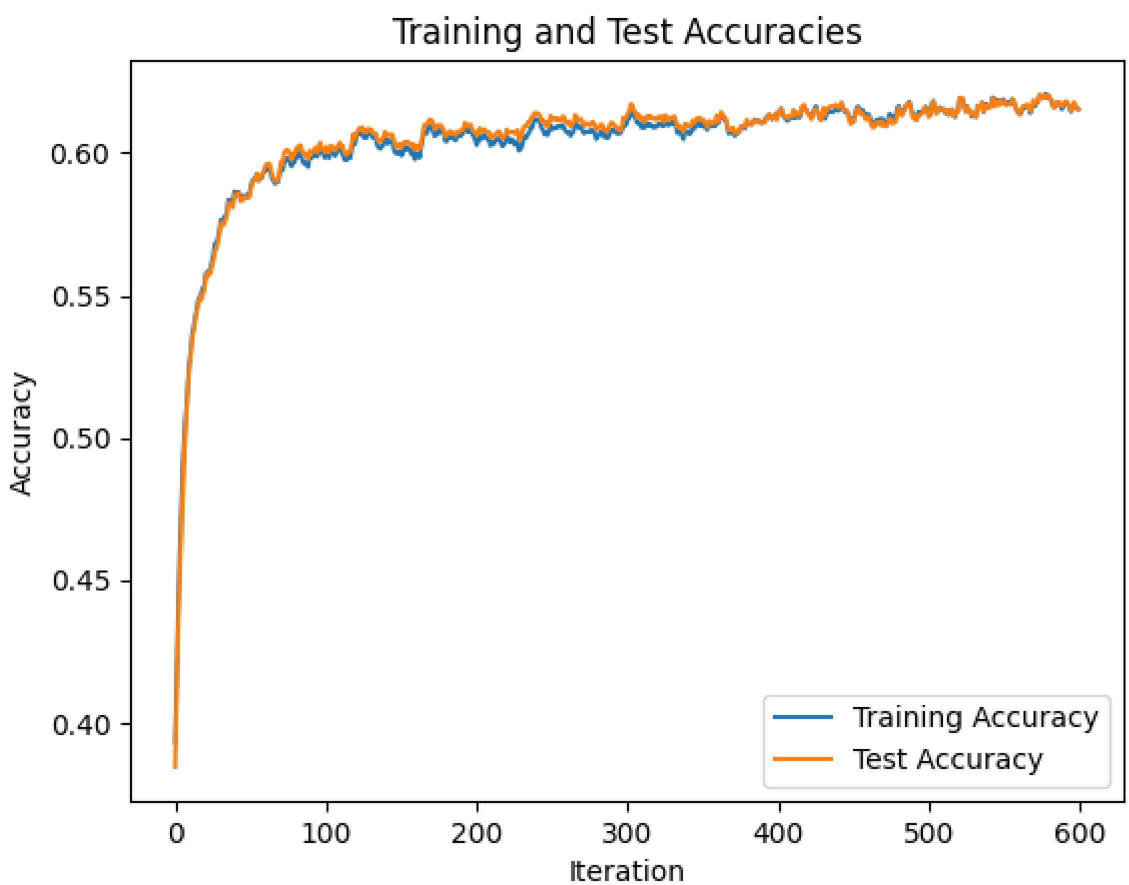
```
# Define forward and backward pass with quadratic loss
def forward_backward_with_quadratic_loss(X, y, v, W):
    # Forward pass
    h = relu(np.dot(X, W.T))
    y_pred = np.dot(h, v)
    # Compute loss
    loss = quadratic_loss(y, y_pred)
    # Backward pass
    delta = quadratic_loss_derivative(y, y_pred)
    grad_v = np.dot(delta, h) / len(X) + reg_lambda * v
    grad_W = np.dot((np.outer(delta, v) * relu_derivative(np.dot(X, W.T))).T, X) / ler
    return loss, grad_v, grad_W


def train(X_train,W,v):
        h_train = relu(np.dot(X_train, W.T))
        y_train_pred = np.dot(h_train, v)
        y_train_pred_binary = np.where(y_train_pred > 0, 1, 0)
        train_acc = np.mean(y_train_pred_binary == y_train)
        return train_acc
def test(x_val,W,v):
        h_val = relu(np.dot(X_val, W.T))
        y_val_pred = np.dot(h_val, v)
        y_val_pred_binary = np.where(y_val_pred > 0, 1, 0)
        val_acc = np.mean(y_val_pred_binary == y_val)
        return val_acc
```

```
import numpy as np
train_value_q=[]
test_value_q=[]
# Train and evaluate the model for each value of k
for k in [5, 40, 200]:
```

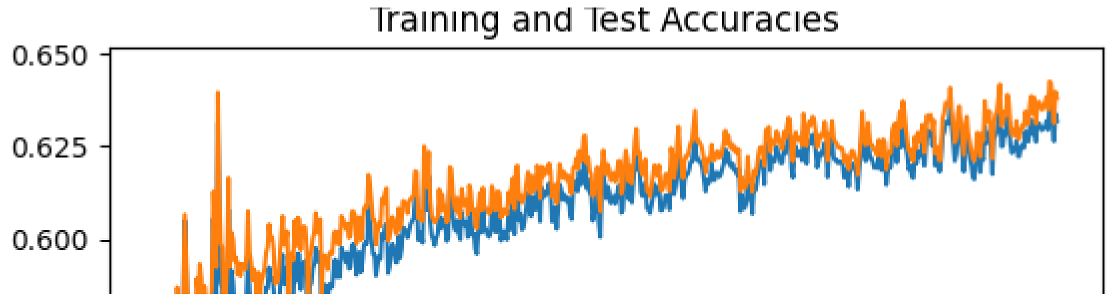
```
    # Initialize the neural net
    W = np.random.normal(0, 1 / np.sqrt(d), (k, d))
    v = np.random.normal(0, 1 / np.sqrt(k), k)
    # Train the model
    for epoch in range(epochs):
        X_train, y_train = shuffle(X_train, y_train)
        for i in range(0, X_train.shape[0], batch_size):
            X_batch = X_train[i:i+batch_size]
            y_batch = y_train[i:i+batch_size]
            # Compute loss and gradients
            loss, grad_v, grad_W = forward_backward_with_quadratic_loss(X_batch, y_bat
            # Update weights
            v -= learning_rate * grad_v
            W -= learning_rate * grad_W
            if i%1000==0:
              train_value_q.append(train(X_train,W,v))
              test_value_q.append(test(X_val,W,v))
    # Test the model
    qtest_acc=test(X_val,W,v)
    print(f"k = {k}, Test accuracy: {qtest_acc}")
    plt.plot(train_value_q, label='Training Accuracy')
    plt.plot(test_value_q, label='Test Accuracy')

# Add labels, title, and legend
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.title('Training and Test Accuracies')
    plt.legend()

# Show the plot
    plt.show()
    train_value_q.clear()
    test_value_q.clear()
```

```
k = 5, Test accuracy: 0.5648
```


```
k = 40, Test accuracy: 0.6173
```




```
k = 200, Test accuracy: 0.6335
```




The simplicity of optimization and the precision of the model can be significantly influenced by the number of hidden units (k) in a neural network. Model complexity and generalizability are frequently traded off when deciding on the number of hidden units.

### Low number of hidden units (k=5):56.48%

When there are fewer hidden units, optimizing becomes simpler. However, due to the model's potential inability to fully represent the underlying structure of the data, this simplicity can result in underfitting. Test accuracy may suffer as a result.

### Moderate number of hidden units (k=40):61.73%

Model complexity and generalizability can be balanced with a reasonable amount of hidden units. As a result, both the training and test sets' accuracy can be enhanced through greater optimization. It frequently represents a reasonable middle ground between underfitting and overfitting, resulting in improved performance as a whole.

### High number of hidden units (k=200):63.35%

The model grows more complex as the number of hidden units rises, which makes it more difficult to optimize due to the rising number of parameters. The model may be able to fit the training data very well, which would result in high training accuracy, but it may not generalize well to new, untested data, which would lead to overfitting and lower test accuracy.

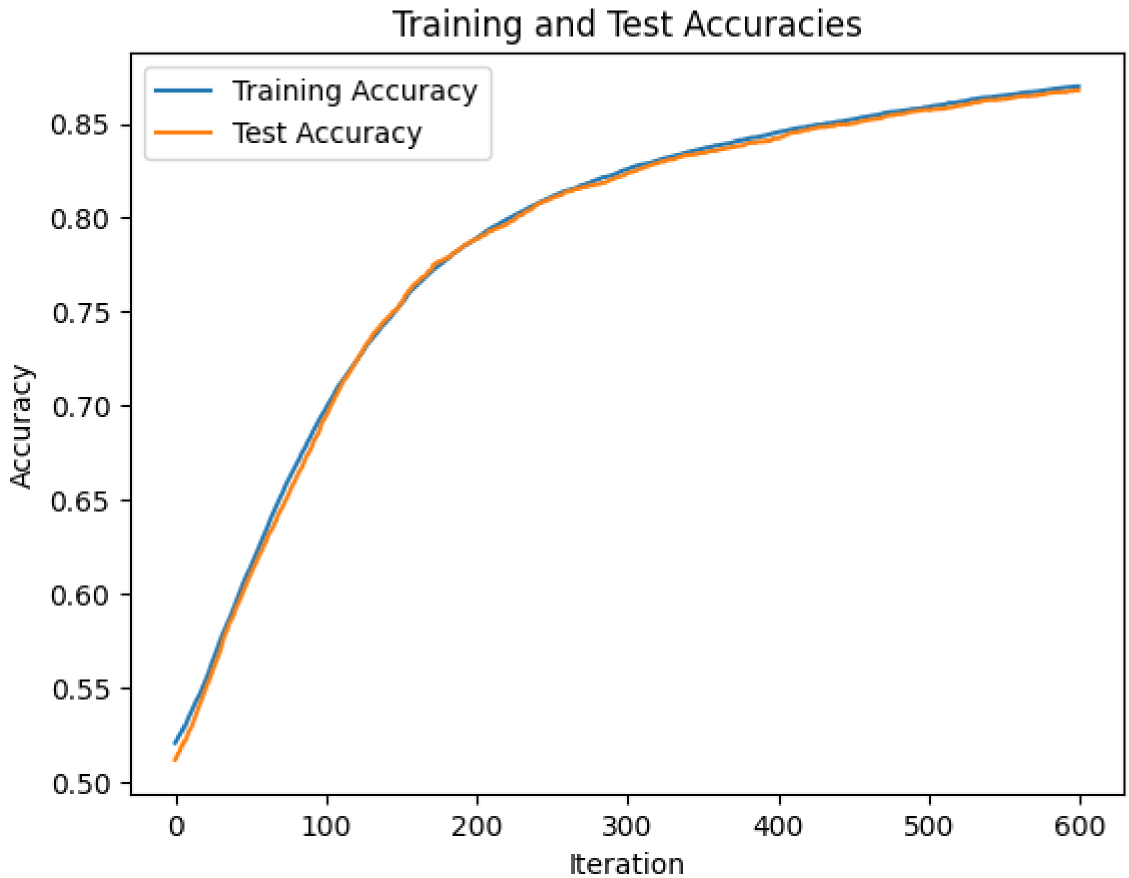
## 4) Neural network classifier with logistic loss

```
# Define forward and backward pass with logistic loss
def forward_backward_with_logistic_loss(X, y, v,W):
    # Forward pass
    h = relu(np.dot(X, W.T))
    y_pred = np.dot(h, v)
    # Compute loss
    loss = logistic_loss(y, y_pred)
    # Backward pass
    delta = logistic_loss_derivative(y, y_pred)
    grad_v = np.dot(delta, h) / len(X) + reg_lambda * v
    grad_W = np.dot((np.outer(delta, v) * relu_derivative(h)).T, X) / len(X) + reg_lar
    return loss, grad_v, grad_W
```

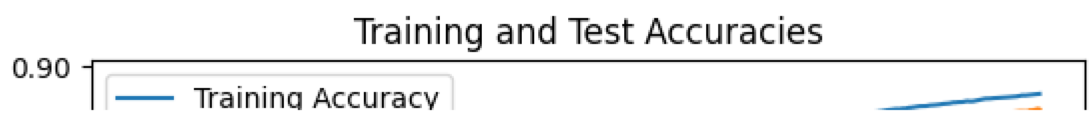
```
def train_l(X_train,W,v):
        h_train = relu(np.dot(X_train, W.T))
        y_train_pred = np.dot(h_train, v)
        y_train_pred_binary = np.where(y_train_pred > 0.5, 1, 0)
        train_acc = np.mean(y_train_pred_binary == y_train)
        return train_acc
def test_l(x_val,W,v):
        h_val = relu(np.dot(X_val, W.T))
        y_val_pred = np.dot(h_val, v)
        y_val_pred_binary = np.where(y_val_pred > 0.5, 1, 0)
        val_acc = np.mean(y_val_pred_binary == y_val)
        return val_acc


train_value_l=[]
test_value_l=[]
# Train and evaluate the model for each value of k
for k in [5, 40, 200]:
    d = 785

    W = np.random.normal(0, 1 / np.sqrt(d), (k, d))
    v = np.random.normal(0, 1 / np.sqrt(k), k)
    # Train the model
    for epoch in range(epochs):
        X_train, y_train = shuffle(X_train, y_train)
        for i in range(0, X_train.shape[0], batch_size):
            X_batch = X_train[i:i+batch_size]
            y_batch = y_train[i:i+batch_size]

            # Compute loss and gradients
            loss, grad_v, grad_W = forward_backward_with_logistic_loss(X_batch, y_batc

            # Update weights
            v -= learning_rate * grad_v
            W -= learning_rate * grad_W
            if i%1000==0:
              train_value_l.append(train_l(X_train,W,v))
              test_value_l.append(test_l(X_val,W,v))
    #Test the model
    ltest_acc=test_l(X_val,W,v)
    print(f"k = {k}, Test accuracy: {ltest_acc}")
    plt.plot(train_value_l, label='Training Accuracy')
    plt.plot(test_value_l, label='Test Accuracy')

# Add labels, title, and legend
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.title('Training and Test Accuracies')
    plt.legend()

# Show the plot
```

```
    plt.show()
    train_value_l.clear()
    test_value_l.clear()
```

```
k = 5, Test accuracy: 0.8679
```




```
k = 40, Test accuracy: 0.8775
```




The simplicity of optimization and the precision of the model can be significantly influenced by the number of hidden units (k) in a neural network. Model complexity and generalizability are frequently traded off when deciding on the number of hidden units.

## Low number of hidden units (k=5):86.79%

When there are fewer hidden units, optimizing becomes simpler. However, due to the model's potential inability to fully represent the underlying structure of the data, this simplicity can result in underfitting. Test accuracy may suffer as a result.

## Moderate number of hidden units (k=40):87.75%

Model complexity and generalizability can be balanced with a reasonable amount of hidden units. As a result, both the training and test sets' accuracy can be enhanced through greater optimization. It frequently represents a reasonable middle ground between underfitting and overfitting, resulting in improved performance as a whole.

## High number of hidden units (k=200):89.12%

The model grows more complex as the number of hidden units rises, which makes it more difficult to optimize due to the rising number of parameters. The model may be able to fit the training data very well, which would result in high training accuracy, but it may not generalize well to new, untested data, which would lead to overfitting and lower test accuracy.

## 5) Linear Models vs Neural Networks:

Using a linear combination of input features, linear models are a family of straightforward models that generate predictions. They are comparatively simple to comprehend, analyze, and optimize. However, because of their simplicity, they might not be able to identify intricate data patterns, which would reduce their accuracy for some applications.

While learning complicated, non-linear correlations between input characteristics and target outputs, neural networks are a class of models that are made up of interconnected layers of neurons. While neural networks can perform with more precision on some tasks, their complexity can make them more difficult to optimize and comprehend.

## Comparing the performance of the given models:

Linear model accuracy: 89.87%
Neural net with quadratic loss:

k=5 The Accuracy :56.48%
k=5 The Accuracy :61.73%
k=5 The Accuracy :63.35%

Neural net with logistic loss:

k=5 The Accuracy :86.79%
k=5 The Accuracy :87.75%
k=5 The Accuracy :89.12%

In this situation, the accuracy of the linear model and the neural network with logistic loss are comparable, whereas the accuracy of the neural network with quadratic loss is noticeably inferior. As a classification task, the presented problem is likely one, and logistic loss is a preferable method for handling it. The quadratic loss may be more susceptible to outliers and does not directly optimize the probabilities, making it unsuitable for classification applications. The performance of a model in terms of optimization and test/train accuracy can be considerably impacted by the selection of the proper loss function.